NPS-CS-08-004

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Forensic Capabilities For Service-Oriented Architectures

By

J. B. Michael, M. Shing, and D. Wijesekera

February 25, 2008

Approved for public release; distribution is unlimited

Prepared for: PEO C4I & Space, PMW 180
4301 Pacific Highway
San Diego, CA 92110-3217

THIS PAGE INTENTIONALLY LEFT BLANK

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

Daniel T. Oliver
President

Leonard A. Ferrari
Provost

This report was prepared for and funded by the PEO C4I & Space, PMW 180, Intelligence, Surveillance and Reconnaissance and Information Operations (ISR/IO).

This report was prepared by:

_____
James Bret Michael
Professor of Computer Science and Electrical and Computer Engineering
Naval Postgraduate School

Reviewed by:

_____
Peter J. Denning, Chairman
Department of Computer Science

Released by:

_____
Dan C. Boger
Interim Associate Provost and
Dean of Research

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** February 25, 2008	**3. REPORT TYPE AND DATES COVERED** Technical Report	
4. TITLE AND SUBTITLE: Title (Mix case letters) **Forensic Capabilities For Service-Oriented Architectures**			**5. FUNDING NUMBERS** N6600107WR00222
6. AUTHOR(S) J. B. Michael, M. Shing, and D. Wijesekera			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-08-004
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) PEO C4I & Space, PMW 180, 4301 Pacific Highway, San Diego, CA 92110-3217			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			**12b. DISTRIBUTION CODE**
13. ABSTRACT (maximum 200 words) This report describes a framework to provide on-line forensic capabilities to service oriented architecture via Forensic Web Services (FWS) and runtime execution monitoring. The FWS is a new type of web services to be used by other web services (of an independent agency) to securely maintain transactional records of interest between other web services. The framework uses runtime execution monitoring to search the transactional log for interesting (or suspicious) service invocation sequences to recreate non-repudiable evidence of transactional history for use in a court of law.			
14. SUBJECT TERMS Service oriented architecture, forensic web services, runtime execution monitoring			**15. NUMBER OF PAGES** 23
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

1. Introduction

Large systems-of-systems (SoSes) are typically made up of a federation of existing systems and developing systems interacting with each other over a network to provide an enhanced capability greater than that of any of the individual systems within the system-of-systems. Service-oriented architecture (SOA) and the supporting Web Services (WS) technology hold promise to create SoSes that are interoperable, composable, extensible, and dynamically reconfigurable. The DOD has mandated the basic WS framework standards to be used in the development of its services for use in the Global Information Grid (GIG) and Network Centric Enterprise Services (NCES) programs. Service-level compositional techniques such as choreography, orchestration, dynamic invocation, and brokering, are used to create complex dependencies between web services belonging to different organizations. These services, however, can be exploited by rogue users when the services have localized or compositional flaws. Investigating incidents of misuse of web services requires that dependencies between service invocations be retained in a neutral and secure manner so that the alleged activity can be recreated in an undeniable way while preserving evidence that could lead to and support appropriate prosecutorial activity. Material evidence currently extractable from web servers such as log records and XML firewall alerts from end-point services do not have forensic value because defendants can rightfully claim that they did not send that message and that the plaintiff fabricated or altered the log record to deceive the court. In order to facilitate and base such investigations on reliable infrastructure that can convince judicial systems, Wijesekera *et al.* propose designing *Forensic Web Services (FWS)* that preserve appropriate evidence to recreate the composed web service invocations independent of the parties with a vested interest in the transactional messages [1].

This report describes a framework to provide on-line forensic capabilities to service oriented architecture via FWS and runtime execution monitoring. Section 2 lists the requirements of FWS. Section 3 summarizes the FWS proposed by Wijesekera *et al.* Section 4 describes the use of runtime execution monitoring to examine the transactional evidence for complex transaction scenarios involving multiple web services.

2. Forensic Web Services Requirements

In this report, we address three high-level requirements for forensic web services.

(1) Trusted third party over a secure and reliable environment

It is essential that the forensic data are collected and processed by an independent, trusted third party. We can build upon the extensive research on Trusted Third Party (TTP) protocols to establish non-repudiation of the data collected by the FWSes. Moreover, the FWSes should run over a secure network layer that provides:

(a) authentication of all parties involved,

(b) confidentiality and integrity of the communication channels, and

(c) reliable messaging over the communication channels.

(2) Pair-wise evidence logging with time stamping

The essential task of the FWS is to collect evidence of transactions that occur between pairs of requester WS and server WS at the time of invoking the service. All transactional evidence collected by the FWS must be time-stamped to include:

(a) service request time – when the requester sends a message to the server according to the requester's clock,

(b) service response time – when the server sends the reply to the requester according to the server's clock,

(c) service request time-out – when the FWS sends the requester an attestation to the server's failure to respond to the service request within the time allowed according to the FWS's clock,

(d) server availability time – when the FWS sends the server an attestation to the server's availability according to the FWS's clock.

(3) Comprehensive evidence generation

On demand, the FWS will, in collaboration with other FWSes, compose transactional history of complex transaction scenarios involving multiple web services that occurred during specific periods and met specific transactional patterns.

3. The Forensic Web Services Framework

The FWS framework is made up of a set of collaborating FWSes, as illustrated in Figure 1. To access the services of a registered FWS system, a web service queries the FWS registry and then uses the location information to register with the FWS. Any client requesting services of a web service must re-route its transactional messages through its FWS agent (called the *operator FWS*), which acts as a Trusted Third Party (TTP) that monitors the service requests (and corresponding responses) involving its client. For example, the web services WS-A, WS-B and WS-C have selected, respectively, FWS-1, FWS-2 and FWS-3 as their operator FWS in Figure 1. The following are necessary for FWS systems to function as required:

(1) There should be a message format for communicating WS-Forensics layer messages and storing them in the FWS servers.

(2) All web services must re-route their transactional messages through FWS servers.

(3) The WS call stack must be enhanced with a WS-Forensics layer. (See Section 3.4 for details.)

(4) The underlying system must provide a trust base and cryptographic services.

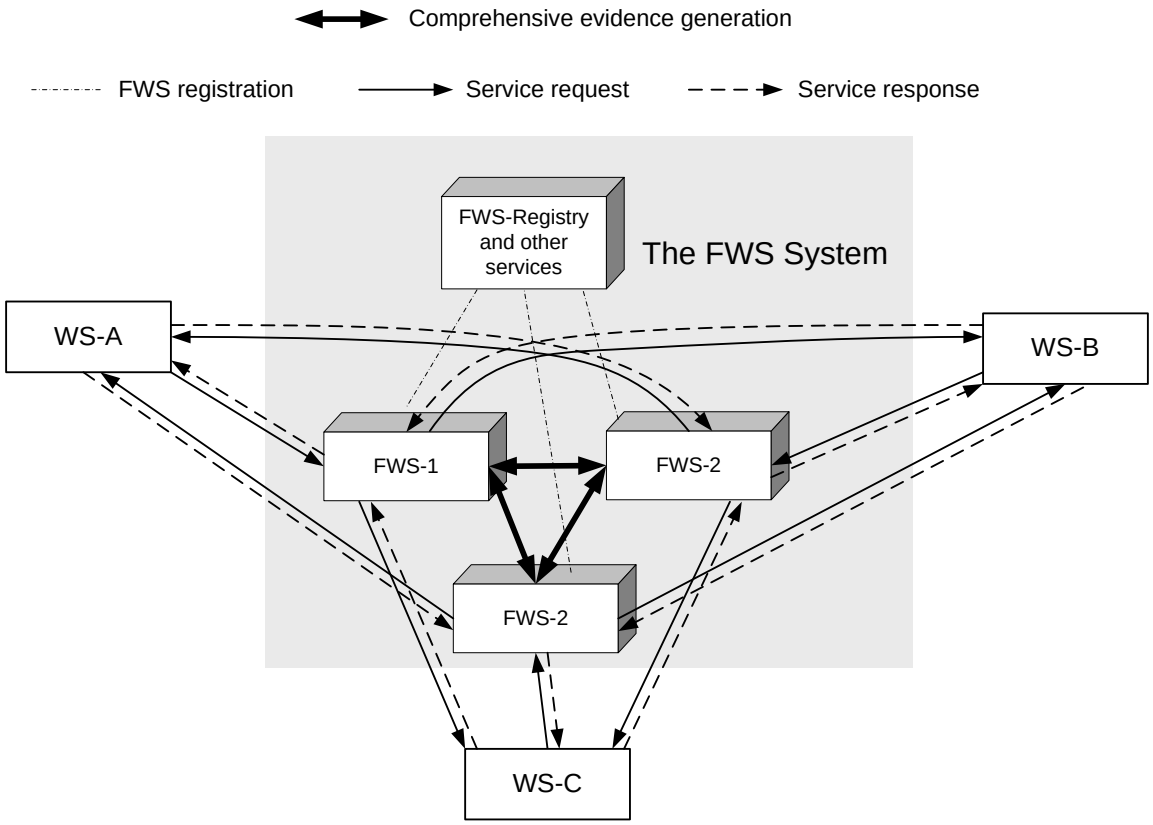


Figure 1. The FWS Framework

3.1 Format for the WS-Forensics Messages

WS-Forensics uses the message format of <#session|#message|#ds:Signature$_K$(#session|#message/sequence|#message/envelope)> to exchange between sending WS, FWS and receiving WS. Here the session element identifies a WS-Forensics conversation, and message corresponds to an element carrying the actual upper layer message along with its sequence number (message/sequence) in the conversation. For instance, sequence number 2 corresponds to a response message if message exchange pattern type (MEPType) is two-way and the protocol is the *Simple Evidence Layer Protocol (SELP)* [2]. At each endpoint, either the sender or the receiver signs the session, message/sequence, and message/envelope parts of the message in the ds:Signature element of the message. Listing 1 shows a sample message instance transmitted in this format.

```
…
<soap:Body>
<p1:fwsMessage ...>
    <p1:session id="session" protocol="#SELP" >
        <p1:sessionID algorithm="URI">
            <p1:id>uuid:21213131313232323222</p1:id>
        </p1:sessionID>
        <p1:MEPType>Two-Way</p1:MEPType>
        <p1:agreement>
            <p1:agreementID algorithm="URI">
```

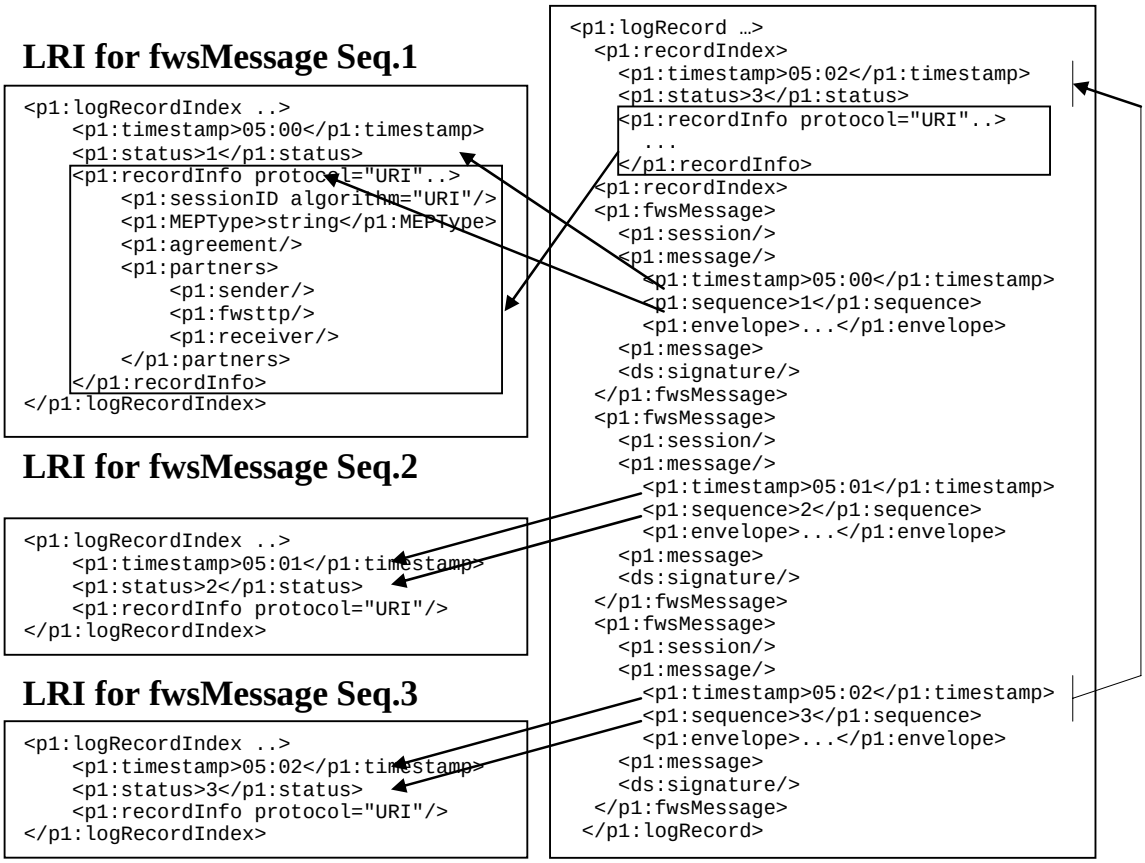
```
                <p1:id>www.contracts.com/#231322323123132132</p1:id>
            </p1:agreementID>
        </p1:agreement>
        <p1:partners>
            <p1:sender> //www.portalservices.com
            <p1:fwsttp> //fws-2.forensicwebservice.com
            <p1:receiver> //www.weatherservices.com
        </p1:partners>
    </p1:session>
    <p1:message >
        <p1:timestamp>2002-10-10T12:00:00-05:00</p1:timestamp>
        <p1:sequence id="sequence">1 </p1:sequence>
        <p1:envelope id="envelope">$EnvelopeFromUpperLayer$</p1:envelope>
    </p1:message>
    <p2:Signature>
        <p2:SignedInfo>
            <p2:Reference URI="#session" >
            <p2:Reference URI="#sequence" >
            <p2:Reference URI="#envelope" >
        </p2:SignedInfo>
        <p2:SignatureValue>
        <p2:KeyInfo>
    </p2:Signature>
</p1:fwsMessage>
</soap:Body>
```

Listing 1: A Sample FWSMessage [1]

3.2 WS-Forensics Messages Recording

WS-Forensics FWS stores the messages in two formats, LogRecordIndex (LRI) and LogRecord (LR), as shown in Listing 2. A LRI refers to the record of a single fwsMessage within a WS-Forensics conversation. LR stores entire WS-Forensics sessions including all fwsMessages delivered to and/or generated by the FWS. LRI records are used for two purposes: (1) for quick searches and (2) for keeping track of the location of the entire LR. Each LRI is stored at both FWSes (*operator* and *non-operator FWS*). LR, on the other hand, is stored only at the *operator FWS* and can be reached using the LRIs that refer to it.

A FWS storing a LRI sets the value of its status field to that of the message/sequence part of the fwsMessage. The FWS also sets the timestamp with the value of message/timestamp part of the fwsMessage and the recordinfo with the value of session part of the fwsMessage. The envelope and ds:signature parts are not represented in LRIs but in LRs. LR contains the recordIndex part that has the final timestamp and status values of the conversation to timestamp and sequence values of the last fwsMessage in the conversation, respectively.



Listing 2: Sample LRI and LR records [1]

3.3 WS-Forensics Messages Routing

Routing transactional information through FWS servers requires that all transactions be reliably intercepted and routed. As stated, FWS servers gather pair-wise transactional evidence that flow between sender and receiver web services, using the *Simple Evidence Layer Protocol (SELP)* [2]. There are four entities involved in the process: *sender*, *receiver*, *operator FWS*, and *non-operator FWS*. *Operator FWS* refers to a FWS selected by either party to manage the steps listed below (illustrated pictorially in Figures 2 and 3), and the *Non-operator FWS* belongs to the other party.

(1) FWS receives MsgSeq.1 (<#session|#message|#ds:Signature$_{\text{Sender-K}}$(#session|“1”|#env)>).

(2) Validates, stores the message, creates an LR and LRI for MsgSeq.1 and notify non-operator FWS.

(3) MsgSeq.1 is forwarded to the Receiver and starts a timer.

(4) If the response MsgSeq.2 cannot reach the FWS before timing out then MsgSeq.-1 (<#session|#message|#ds:Signature$_{\text{FWS-K}}$(#session|“-1”|#env)>) is signed by the FWS; it is stored and sent back to the Sender and an LRI is created and sent to the non-operator FWS.

If MsgSeq.2 (<#session|#message|#ds:Signature$_{Receiver\text{-}K}$(#session|"2"|#env)>) arrives on time and passes the contractual validity test, it is forwarded to the sender and stored in FWS along with notifying the non-operator FWS with its LRI.

If MsgSeq.2 fails the contractual validity test, then MsgSeq.-2 (<#session|#message|#ds:Signature$_{FWS\text{-}K}$(#session|"-2"|#env)>) is signed by the FWS; it is stored and sent back to the Sender and an LRI is created and sent to the non-operator FWS.

(5) FWS creates, signs and sends MsgSeq.3 (<#session|#message|#ds:Signature$_{FWS\text{-}K}$ (#session|"3"|#env)>) to the Receiver. It also stores the message in the LR and sends the LRI to the non-operator FWS.

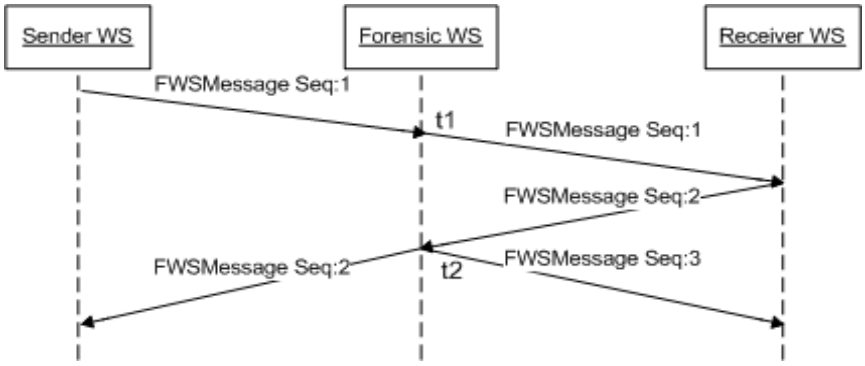


Figure 2. An Operator FWS managing the SELP protocol [1]

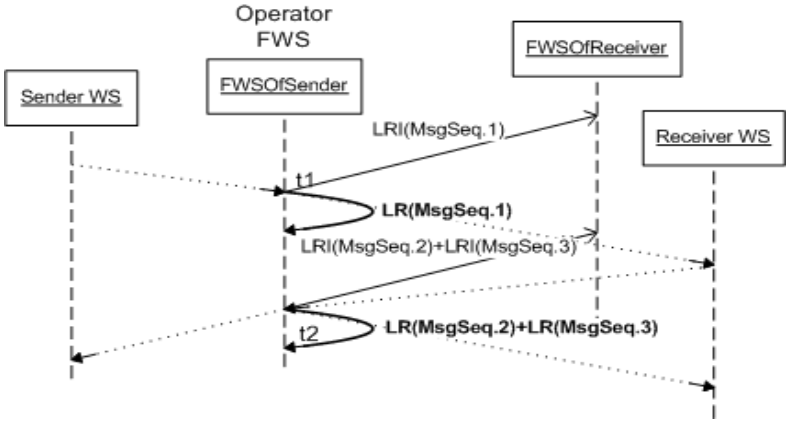


Figure 3. An Operator FWS storing messages [1]

3.4 Enhanced Web-Services Call Stack

The existing WS call stack consists of a three layers: The bottom layer consists of the SOAP[1] messages; the middle layer consists of WS-Secure Conversations; and the top layer consists of the Web Services Description Language (WSDL) specifications. SOAP and WSDL are part of the basic WS framework standards. SOAP provides the standard language for messaging format used by the service and its requestor, while WSDL provides the standard language for describing the point of contact for a service provider (a.k.a. the service endpoint or just endpoint), the public interface of an endpoint (i.e., the way the requestors should communicate with the service provider), and the physical address of the service.

Wijesekera *et al.* propose to add a forensic layer in between the middle layer and the top layer to reroute transactions through the FWS servers (Figure 4), and have a sender process and a receiver process sitting in front of each web service endpoint (Figure 5).

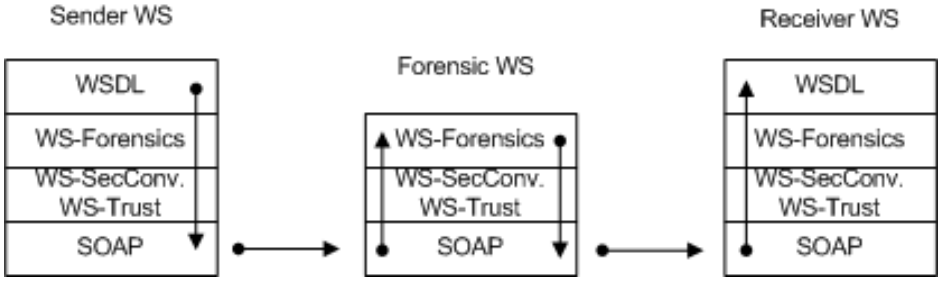


Figure 4. The enhanced WS call-stack [1]

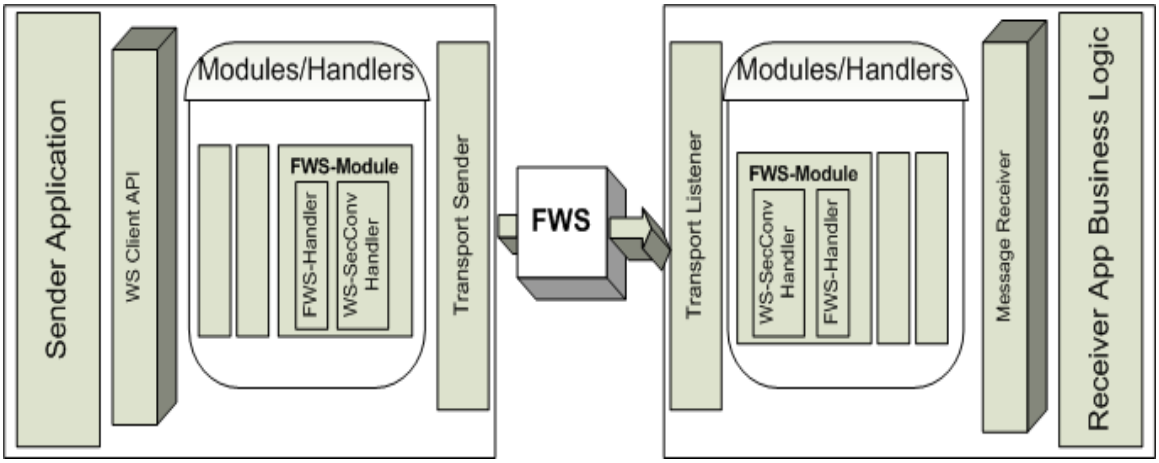


Figure 5. FWS-Handler Module Architecture (adapted from [3])

[1] SOAP initially stood for Simple Object Access Protocol. When W3C adopted SOAP as a standard, the acronym was considered misleading and therefore dropped in favor of just SOAP.

The Sender Process FWS-Handler captures the SOAP message from the upper layer and encapsulates the message in the WS-Forensics message format by adding signatures, routing the message to the operator FWS and so on, and submitting the result to the WS-SecureConversation/WS-Trust handlers shown in Figure 4.

The Receiver Process: FWS-Handler handles the WS-Forensics fwsMessage from the lower layer. After validating the signature according to the WS-Forensics session context the handler extracts the original SOAP message and either passes it to another handler (if such handler exists) in the chain or dispatches it to the intended service|portype|operation entity.

3.5 Security Requirements for Underlying Layer

WS-Forensics is designed to run over a secure layer with following services: (a) authentication of all parties involved, (b) confidentiality and integrity of the communication channels, and (c) reliable messaging over the communication channels.

Two properly implemented standards, WS-Trust [4] and WS-SecureConversation [5], satisfy these requirements. WS-Trust issues, renews and verifies tokens to support the verification of message confidentiality, integrity, authentication, and so on. WS-SecureConversation builds secure sessions using XML encryption and signature.

The processes described in Section 3.4 require secure channels between endpoint web services and FWS nodes. The steps below show how a WS-Forensics message, fwsMessage, traverses from a sender to a FWS and subsequently to a receiver.

(1) WS-SecureConversation/WS-Trust handler of the sender grabs the fwsMessage. The handler then builds a secure conversation by means of the Security Context Token (SCT) obtained from the Security Token Service (STS). FWS nodes also may have this role. The fwsMessage is encrypted by WS-SecureConversation and then pushed into the transport layer to be sent to the FWS node through the conversation

(2) WS-SecureConversation/WS-Trust handler of FWS node receives the encrypted SOAP message, decrypts it, extracts the actual fwsMessage, and pushes the message into the WS-Forensics layer to be processed as described in the next section.

(3) After processing the fwsMessage, the FWS node pushes the message to its WS-SecureConversation/WS-Trust handler to build another secure conversation with the receiver as described in the first step. Then the message is encrypted by the security handler, to be sent to the receiver through the conversation.

(4) WS-SecureConversation/WS-Trust handler of the receiver receives the encrypted SOAP message, decrypts it, extracts the actual fwsMessage, and pushes it into the WS-Forensics layer to be dispatched.

3.6 Pair-wise Evidence

The SELP protocol and FWS event logs retain the evidence to verify the following claims, mapped to messages in Table 1:

- **Evidence of Origin (EOO):** attestation of message send-time and origin.
- **Evidence of Delivery (EOD):** message acceptance by the intended receiver and the acceptance time.
- **Evidence of Failure (EOF):** attestation of message not acknowledged by intended receiver within time allowed.
- **Evidence of Availability (EOA):** attestation of server's availability in a specific time interval.
- **Evidence of Agreement Violation (EOV):** attestation of contractual violation by the server.

Evidence Type	Signer	FWS Implementation
EOO	Sender of message	MsgSeq.1 and MsgSeq.3
EOD	Receiver of message	MsgSeq.1 and MsgSeq.2
EOF	FWS	MsgSeq.-1
EOA	FWS	MsgSeq.0
EOV	FWS	MsgSeq.-2

Table 1: Notation for Evidence Types

4. Transactional Evidence Generation

The creation of comprehensive evidence of a misuse scenario requires the examination of pair-wise transactional evidence stored in multiple FWSes for interesting sequencing behaviors, which are behaviors that consist of sequences of events, conditions and constraints on data values, and timing. In its vanilla form, sequencing behavior specifies sets of legal (or illegal) sequences, such as the following automotive body-logic requirement:

> *Once engine is turned off, compartment lights must be on until driver door is opened.*

Sequencing behavior has two types of common constraints:

(1) Timing constraints – describe the timely start and/or termination of successful or forbidden computations, such as the deadline of a periodic computation or the maximum response time of an event handler. For example,

> *The sqrt() function must complete its computation and return an answer within 200 milliseconds from the time it is called*.

(2) Time-series constraints – describe the timely execution of a sequence of computations within a specific duration of time. For example,

> *Whenever the system load (L) exceeds 75% of the MaxLoad, L must be reduced back to 50% of the MaxLoad within 1 minute and must remain at or below 60% of the MaxLoad for at least 10 minutes.*.

In this section, we describe the use of MSC-Assertions to specify interesting behavior of event sequences and runtime execution monitoring to both examine and construct the transactional evidence for complex transaction scenarios involving multiple web services.

4.1 Expressing Sequencing Behavior as Message Sequence Chart Assertions

MSC Assertions are a formal language extension of UML Message Sequence Charts (MSCes) superimposed with UML statecharts [6]. They have the look and feel of UML MSCes and UML statecharts, yet they are formal and executable. For example, unlike UML MSCes, MSC Assertions provide for distinguishing between events that *can* occur and those that *must* occur. In addition, MSC Assertions are capable of specifying infinite sets of scenarios.

MSC Assertions are based on Statechart diagrams superimposed on MSC diagrams and augmented with Java (or C++) conditions and actions. For example, Figure 6 shows the MSC Assertion for a time-bound requirement of a travel agent service:

> *R1: The travel agent must obtain bids from at least two airlines and two hotels and return a flight and a hotel matching the customer's request within 30 seconds from the time the customer issues his travel request.*

The MSC Assertion of Figure 6 looks, for the most part, like a UML MSC, but it enjoys the following unique features:

(1) An MSC Assertion is written from the standpoint of an *observer*, and can be used for runtime monitoring of the target application. Consider for example the message *reqFlight(Flight f)* sent from the Travel Agent to Airline #1. While a UML MSC might consider an interpretation where this event is *generated* by the Travel Agent, for an MSC Assertion, it is meant that the MSC Assertion should monitor-for, or listen-for, this event flowing from Travel Agent to Airline #1. Note that while the Travel Agent service may send out many requests to different airlines for bids, the MSC Assertion only needs to observe two of such requests to satisfy the requirement R1.

(2) An MSC Assertion *allows loops and transitions back up the vertical task bar*. In Figure 6 for example, the Travel Agent will return to the *Waiting* state if the condition *aBidCount* ≥ 2 *&& hBidCount* ≥ 2 is false. This feature is in contrast to UML MSCes where a vertical task bar represents a *timeline* and where clearly a task cannot move back in time. An MSC Assertion however, considers a vertical task bar as a *progression of states*, like a state diagram drawn vertically. It therefore permits loops.

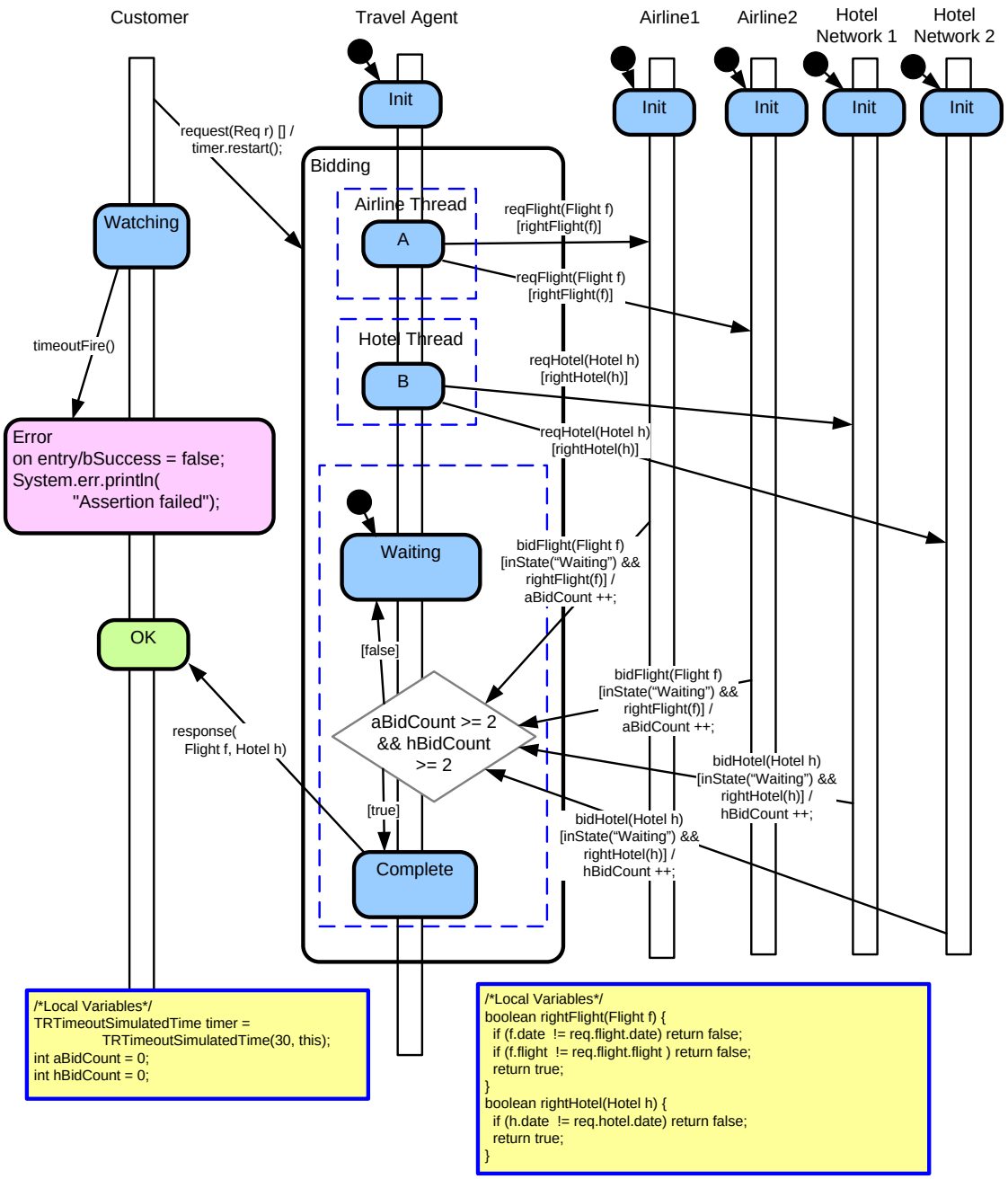


Figure 6. A MSC Assertion for the Travel Agent Service [6]

(3) *States and actions.* As discussed above and as illustrated in Figure 6, an MSC Assertion task might contain both implicit and explicit states. The purpose of explicit states is to specify actions, which are code snippets (written in Java or C++, depending on the code generator chosen) to be performed, such as *aBidCount++* or *rightFlight(Flight h)*. For example, the Customer will remain in its implicit initial state until the event *request(Req r)* is observed leaving the *Customer.* The Customer then enters the *Watching* state. The Customer will remain in *Watching* state until

either the event *response(Flight f, Hotel h)* is observed arriving at the *OK* state, or the timeout event is detected.

(4) *Java/C++ underlying language and code generation*. An MSC Assertion is a diagrammatic representation of a Java or C++ class that implements the requirement as a monitor. Hence, all variables and functions declared in the local-variables boxes of Figure 6 are actually properties of this generated class.

(5) *Parameterized events*. An MSC Assertion event can contain objects as actual parameters. In Figure 6, the transition annotated with the message *bidFlight(Flight f)*, from Airline #1 to the Travel Agent, is sent with some *Flight* object as an argument. Condition guards range over local properties and event arguments (e.g., *rightFlight(f)*).

(6) *An MSC Assertion is an assertion*. It uses the same approach described in [6] for assertion statecharts where it announces a success or failure for every witnessed input scenario. It does so using the built-in *bSuccess* property. The boolean *bSuccess* is true by default. The developer assigns *bSuccess=false* as an action wherever s/he wants the assertion to fail. The JUnit test-case then inspects this property to decide whether a particular test-run failed.

Figure 6 realized requirement R1 as follows. First note that, in the style of the UML MSC notation, the assertion contains six tasks, denoted by the six vertical task bars. Also, the assertion contains local variables *timer*, *aBidCount*, and *hBidCount*, as well as two Boolean functions *rightFlight()* and *rightHotel()* for checking the correctness of the itinerary. The MSC Assertion monitoring starts as a *request(Req r)* event is observed from the Customer task to the Travel Agent task while the Customer task is in its implicit inital state. The 30 second timer is triggered and the Customer task enters its *Watching* state. The Customer will remain in *Watching* state until either the event *response(Flight f, Hotel h)* from the Travel Agent task (while the latter is in its *Complete* state) or the timeout event is detected. If the Customer receives the *response()* message before the timeout event, it will enter the *OK* final state. If the Customer task does not receive the *response()* message before the timeout event, the timeout event will cause the Customer task to enter the *Error* final state; *bSuccess* will be set to false indicating the violation of the requirement.

The Travel Agent task will remain in its *Init* state until it receives the event *request(Req r)*, then it will transition to the *Bidding* state. The *Bidding* state consists of three concurrent threads, in the style of the UML statechart threads [6]. The Travel Agent task will remain in the *Waiting* state until it has received at least two airline bids and two hotel bids. It will then transition to the *Complete* state where the MSC Assertion is ready to observe the event *response(Flight f, Hotel h)* from the Travel Agent task to the Customer task. Clearly, the Travel Agent task must ensure that the bids received indeed satisfy the customer's request. This constraint is manifested as a condition guard *righFlight(f)* or *rightHotel(h)* on the message transition. (*N.B.*: MSC Assertion message transitions have the same *event[guard]/action* look and feel as UML statechart transitions.)

Since we are not interested in the detailed temporal behavior of the Airline tasks and Hotel Network tasks in the requirement R1, we treat these tasks as black boxes. The MSC Assertion only observes the fact that each of these tasks returns a bid to the Travel Agent task only after they have received a request for bid from the Travel Agent task as follows. Each of these four tasks remains in its *Init* state until it receives the request for bid message from the Travel Agent task. It then enters its implicit working state. It will transition from its working state to its implicit terminal state when the MSC Assertion observes that the task returns a bid to the Travel Agent task.

4.2 Runtime Execution Monitoring

Runtime Execution Monitoring of formal specification assertions (REM) is a class of methods for tracking the temporal behavior, often in the form of formal specification assertions, of an underlying application. REM methods range from simple print-statement logging methods to runtime tracking of complex formal requirements (e.g., written in temporal logic) for verification purposes. The National Aeronautics and Space Administration (NASA) used REM for the verification of flight code for the Deep Impact project [7]. In [8], we showed that the use of runtime monitoring and verification of temporal assertions, in tandem with rapid prototyping, helps debug the requirements and identify errors early in the development process. Recently, the Missile Defense Agency (MDA) adopted REM as the primary verification method for the new ballistic missile defense battle manager because of REM's ability to scale as the size and complexity of a system increase, and its support for temporal assertions that include realtime and time-series constraints [9].

4.3 Creating Evidence for Scenarios

As stated, the main objective of the FWS Framework is post-mortem investigations on inter-dependent scenarios containing more than one party in a comprehensive manner, which can be accomplished using the following process:

(1) Define the boundary of scenario generation by specifying the web services being investigated (called *suspected web services*) and the time period of the scenario.

(2) Create a MSC-assertion to describe the sequencing behavior of interest involving the suspected web services. Add calls to exception-handlers in the *Error* flowchart box of the MSC Assertion to collect the event sequence causing the failure whenever the MSC Assertion fails during execution of the web services.

(3) Use the FWS-Registry to locate the FWS Trust Third Parties holding the LR records of any pair-wise transactions involving the suspected web services.

(4) Retrieve the pair-wise transaction evidence for the specified time period from the FWS TTPS.

(5) Re-create the interactions using a discrete event simulator, and use to exercise the MSC-assertion statecharts as runtime execution monitors to collect evidence leading to the failure of the assertions [10].

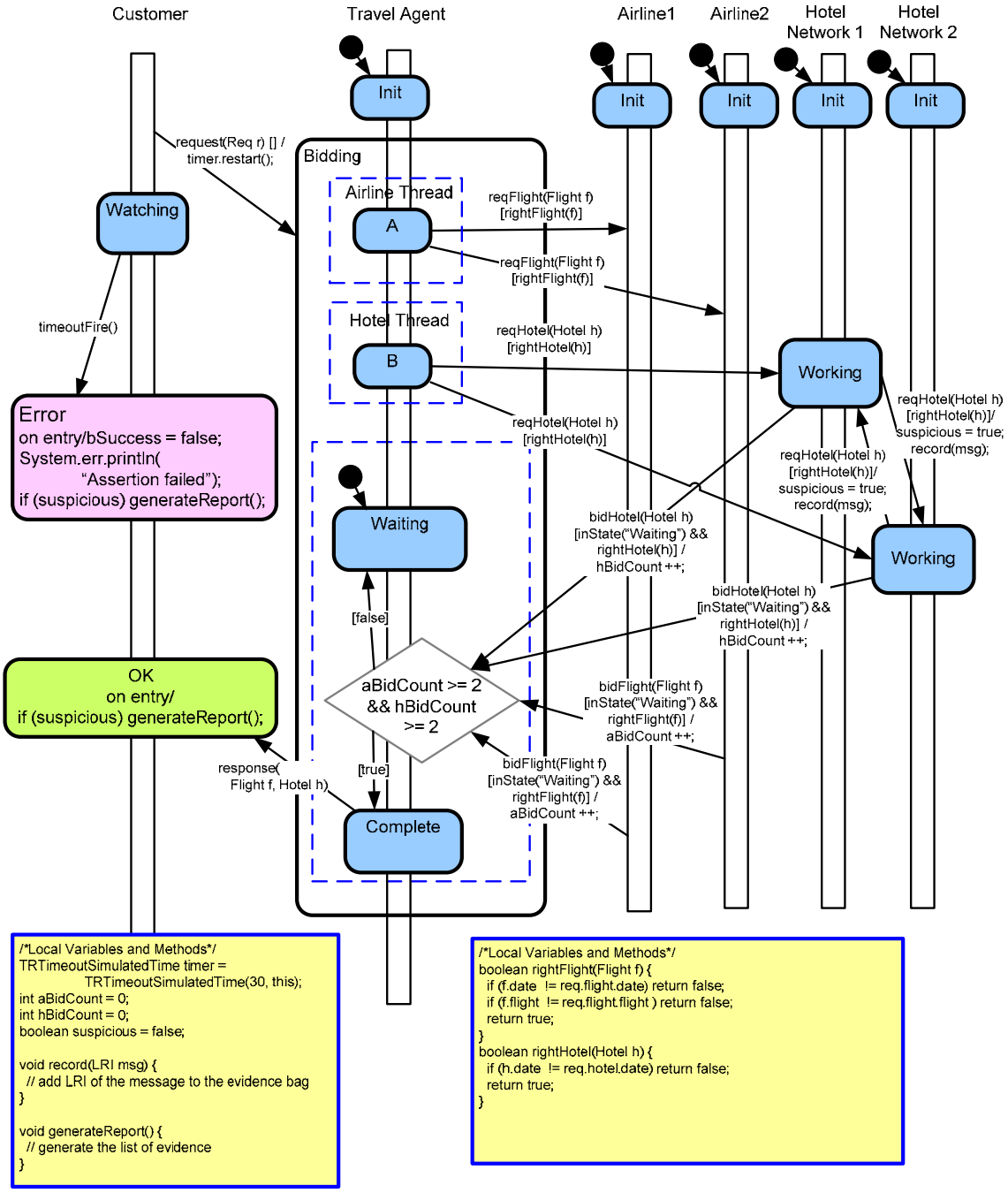


Figure 7. A MSC Assertion for Detecting and Collect Price Fixing Evidence

For example, suppose we suspect that Hotel Networks 1 and 2 are involved in a price fixing scheme and want to collect evidence of such activities. We can create the

MSC-assertion shown in Figure 7 to detect and record all *reqHotel* messages between the two hotel networks from the time when one of the hotel networks received a *reqHotel* message from the Travel Agent to the time when the last of the two hotel networks returned a bid to the Travel Agent. The statechart assertion sets the boolean *suspicious* to true if it detects such communication and reports the evidence when the scenario terminates in either the *OK* state or the *Error* state.

5. Conclusion

In this report, we discussed the need to preserve appropriate evidence to recreate the composed web service invocations independent of the parties with a vested interest in the invocations. We presented a framework to provide this capability as a web service to other web services by logging service invocations at the appropriate level of detail, and the use of MSC Assertions and runtime monitoring to automate the generation of transactional evidence for complex scenarios. Our next step is to develop the necessary software to support the framework and validate the proposed framework with a prototype.

6. References

[1] M. Gunestas, D. Wijesekera and A. Singhal. "Forensic Web Services," in *Proceedings of the Fourth Annual IFIP WG 11.9 International Conference on Digital Forensics*, Kyoto, Japan, January 2008.

[2] A. Herzberg and I. Yoffe, "The Delivery and Evidences Layer," *Cryptology ePrint Archive Report* 2007/139, 2007.

[3] *Apache Axis2 Architecture Guide*, The Apache Software Foundation 2006.

[4] WS-Trust V1.0 Working Draft, OASIS Web Services Secure Exchange TC, http://www.oasisopen.org/committees/download.php/16138/oasis-wssx-ws-trust-1.0.pdf, 2006.

[5] OASIS, WS-SecureConversation 1.3, 2007.

[6] T.S. Cook, D. Drusinsky and M. Shing, "Specification, Validation and Run-time Monitoring of SOA Based System-of-Systems Temporal Behaviors," in *Proceedings of the 2007 IEEE International Conference on System of Systems Engineering*, San Antonio, Texas, April 2007.

[7] D. Drusinsky and G. Watney, "Applying Run-time Monitoring to the Deep-Impact Fault Protection Engine," in *Proc. 28th NASA Goddard Software Engineering Workshop*, Greenbelt, Md., pp. 127-133, Dec. 2003.

[8] D. Drusinsky and M. Shing, "Verification of Timing Properties in Rapid System Prototyping," in *Proc. 14th IEEE International Workshop in Rapid Systems Prototyping*, San Diego, Calif., pp. 47-53, June 2003.

[9] D. Caffall, T. Cook, D. Drusinsky, B. Michael, M. Shing and N. Sklavounos, Formal Specification and Run-time Monitoring within the Ballistic Missile Defense Project, Tech. Report NPS-CS-05-007, Naval Postgraduate School, Monterey, Calif., June 2005.

[10] D. Drusinsky and M. Shing, "Verifying Distributed Protocols using MSC-Assertions, Run-time Monitoring, and Automatic Test Generation," in *Proceedings of the 18th IEEE/IFIP International Workshop on Rapid Systems Prototyping*, Porto Alegre, Brazil, pp. 82-88, June 2007.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52
   Naval Postgraduate School
   Monterey, CA

3. Research Office, Code 09
   Naval Postgraduate School
   Monterey, CA

4. Mr. John J. Shea
   PEO C4I & Space, PMW 180
   San Diego, CA

5. Dr. Bret Michael
   Naval Postgraduate School
   Monterey, CA

6. Dr. Duminda Wijesekera
   George Mason University
   Fairfax, VA

7. Dr. Man-Tak Shing
   Naval Postgraduate School
   Monterey, CA

8. CDR Kurt Rothenhaus
   PEO C4I & Space, PMW 150
   San Diego, CA

9. LTC Thomas Cook
   Naval Postgraduate School
   Monterey, CA

10. Lt. Col. Carl Oros
    Naval Postgraduate School
    Monterey, CA